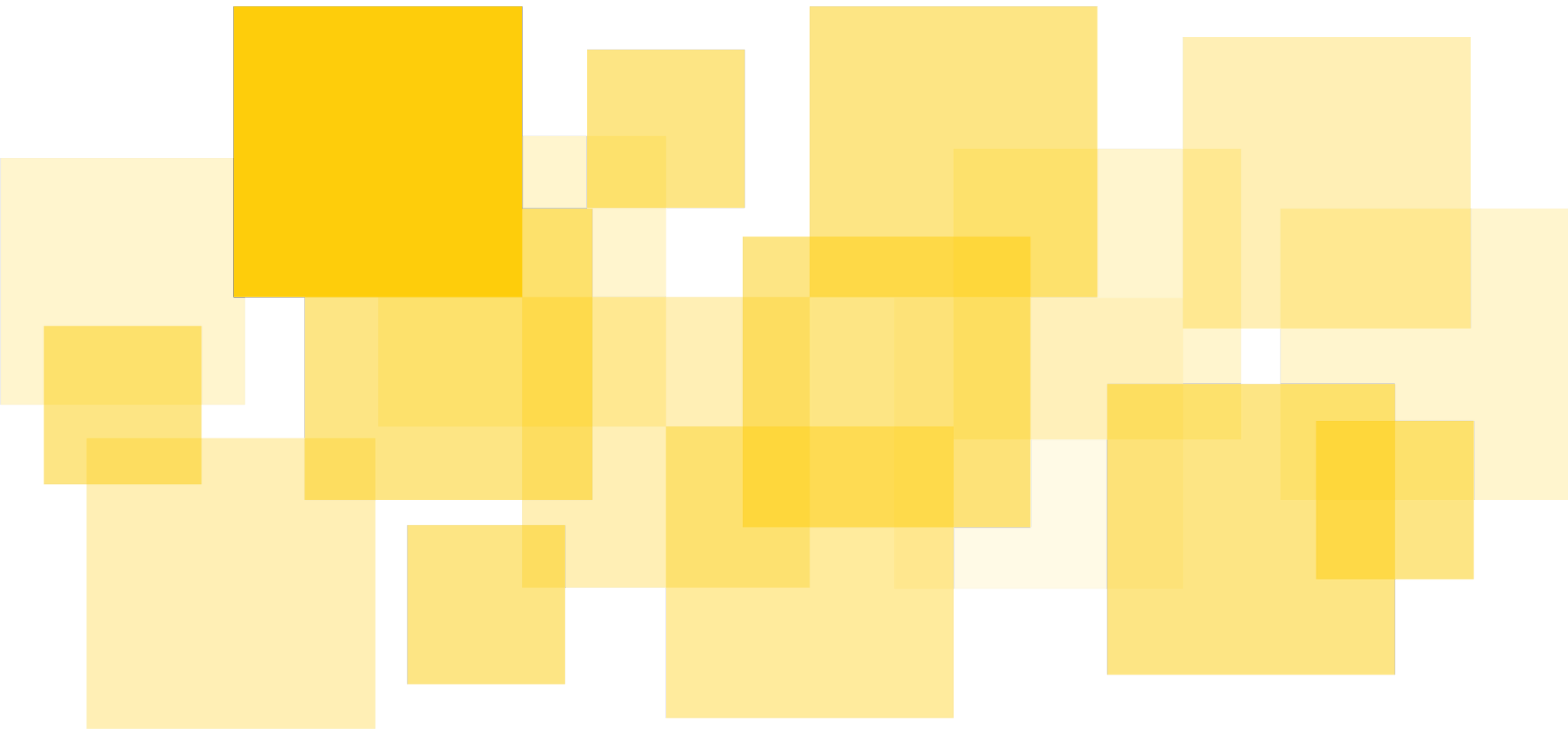
Audit Report

Zivoe - Locker Contracts

Delivered: 2023-08-18



Prepared for Zivoe by Runtime Verification, Inc.

Summary

Runtime Verification, Inc. audited the smart contract source code of the Zivoe project. The comprehensive review was conducted from 17-07-2023 to 21-08-2023.

Zivoe engaged Runtime Verification in checking the security of their Zivoe project, which in its v1, is an international decentralized credit protocol that facilitates on-chain stablecoin lending to regulated entities. Yield is generated from RWA's and brought back on-chain to the protocol, where then the utility is provided to the protocol.

We have identified some issues, which may be found in the sections Findings and Informative Findings. Issues addressed by the client are identified accordingly with the relevant fixed commit provided.

Scope

The audited smart contracts are:

- `lockers/OCC/OCC_Modular.sol`
- `lockers/OCE/OCE_ZVE.sol`
- `lockers/OCL/OCL_ZVE.sol`
- `lockers/OCR/OCR_Modular.sol`
- `lockers/OCT/OCT_DAO.sol`
- `lockers/OCT/OCT_YDL.sol`
- `lockers/OCT/OCT_ZVL.sol`
- `lockers/OCY/OCY_Convex_A.sol`
- `lockers/OCY/OCY_Convex_B.sol`
- `lockers/OCY/OCY_OUSD.sol`
- `Utility/ZivoeSwapper.sol`

The audit has focused on the above smart contracts, and has assumed correctness of the libraries and external contracts they make use of. The libraries are widely used and assumed secure and functionally correct.

The review encompassed the `Zivoe/zivoe-core-foundry` private code repository. The code was frozen for review at commit

The review is limited in scope to consider only contract code. Off-chain and client-side portions of the codebase are *not* in the scope of this engagement.

Assumptions

The audit assumes that all addresses assigned a role must be trusted for as long as they hold that role. Apart from the deployer that is responsible to create and deploy the contracts, an `owner`

role (see OpenZeppelin's access control) is present for some of the contracts in the protocol such as `ZivoeGlobals` and `ZivoeDAO`.

The admin addresses of the respective contracts need to be absolutely trusted. A multisig controlled by the Zivoe team will deploy the protocol and start the ITO period. After the ITO period ends, the governance contract operating on a DAO-based model will take over. Furthermore, we assume that the deployers and the governance address take relevant steps to ensure that the state of the deployed contracts remains correct. In addition to setting the correct state, it is also contingent upon governance to maintain a reasonable state. This includes only accepting trustworthy tokens and setting protocol parameters honestly.

Note that the assumptions roughly assume "honesty and competence". However, we will rely less on competence, and point out wherever possible how the contracts could better ensure that unintended mistakes cannot happen.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in Disclaimer, we have used the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. Second, we carefully checked if the code is vulnerable to known security issues and attack vectors. Finally, we met with the Zivoe team to provide feedback and suggested development practices and design improvements.

This report describes the **intended** behavior of the contracts under review, and then outlines issues we have found, both in the intended behavior and in the ways the code differs from it. We also point out lesser concerns, deviations from best practice and any other weaknesses we encounter. Finally, we also give an overview of the important security properties we proved during the course of the review.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Zivoe: Contract Description and Properties

Overview

In Zivoe protocol, Lockers are contracts that are specialized to manage the various operations that can be performed by the protocol operations such as loan lifecycle management, redemption process management, yield farming and integration to external swap protocols.

The concept of a locker has been abstracted by the ZivoeLocker contract that implements common functionality expected to be provided by the locker contracts. These functionality include transferring assets to/from a locker which are called push and pull operations. Assets to be used by the lockers are limited to be either ERC20, ERC721 or ERC1155 compatible tokens.

Lockers also handle integrations to third party protocols, such as 1Inch, Uniswap/SushiSwap and Curve Finance. A specific contract called ZivoeSwapper is implemented to handle the integration to 1Inch protocol. Following subsections discuss the operation and details of each locker.

ZivoeLocker

This contract defines a common interface and default implementations for the actual Zivoe Lockers. Mentioned common interface contains pull and push variations (transfer tokens to/from the locker) over three token types: ERC20, ERC721 and ERC1155. In addition to these pull and push functions there are also access control functions to determine which of the defined functionalities are supported by the contract that implements a locker. This way, when a locker is being developed it is possible to override access control functions to allow invocation of the default implementations or override the push/pull function itself to provide specific functionality.

Following is a summary of the common functions for lockers:

Function(s)	**Definition**
pushToLocker pushToLockerERC721 pushToLockerERC1155	Transfer specified amount of tokens of type ERC20 / ERC721 / ERC1155 from the owner to the locker.
pullFromLocker pullFromLockerERC721 pullFromLockerERC1155	Transfer all of the tokens in the locker's balance of type ERC20 / ERC721 / ERC1155 from the locker to the owner.

pullFromLockerPartial	Transfer specified amount of ERC20 tokens from the locker to the owner.
pushToLockerMulti pushToLockerMultiERC721	Transfer specified amount (provided as parameter in an array) of tokens for each token (provided as parameter in another array) from the owner to the locker. All of the tokens in the provided array are either of type ERC20 or of type ERC721 according to the function being called.
pullFromLockerMulti pullFromLockerMultiERC721	Transfer all of the tokens in the locker's balance for each token (provided as parameter in an array) from the locker to the owner. All of the tokens in the provided array are either of type ERC20 or of type ERC721 according to the function being called.
pullFromLockerMultiPartial	Transfer specified amount (provided as parameter in an array) of ERC20 tokens for each token (provided as parameter in another array) from the locker to the owner.

The contract also contains can*Function*() functions to determine which of the *Function*s in the table provided above are allowed to be invoked. The default implementation of these *Function*s in the table requires can*Function*() to return `true` whereas default implementation of can*Function*() functions return `false`. Therefore, the lockers that want to enable a particular functionality override can*Function*() to return `true` and either use the default implementation provided in this contract or provide its own implementation.

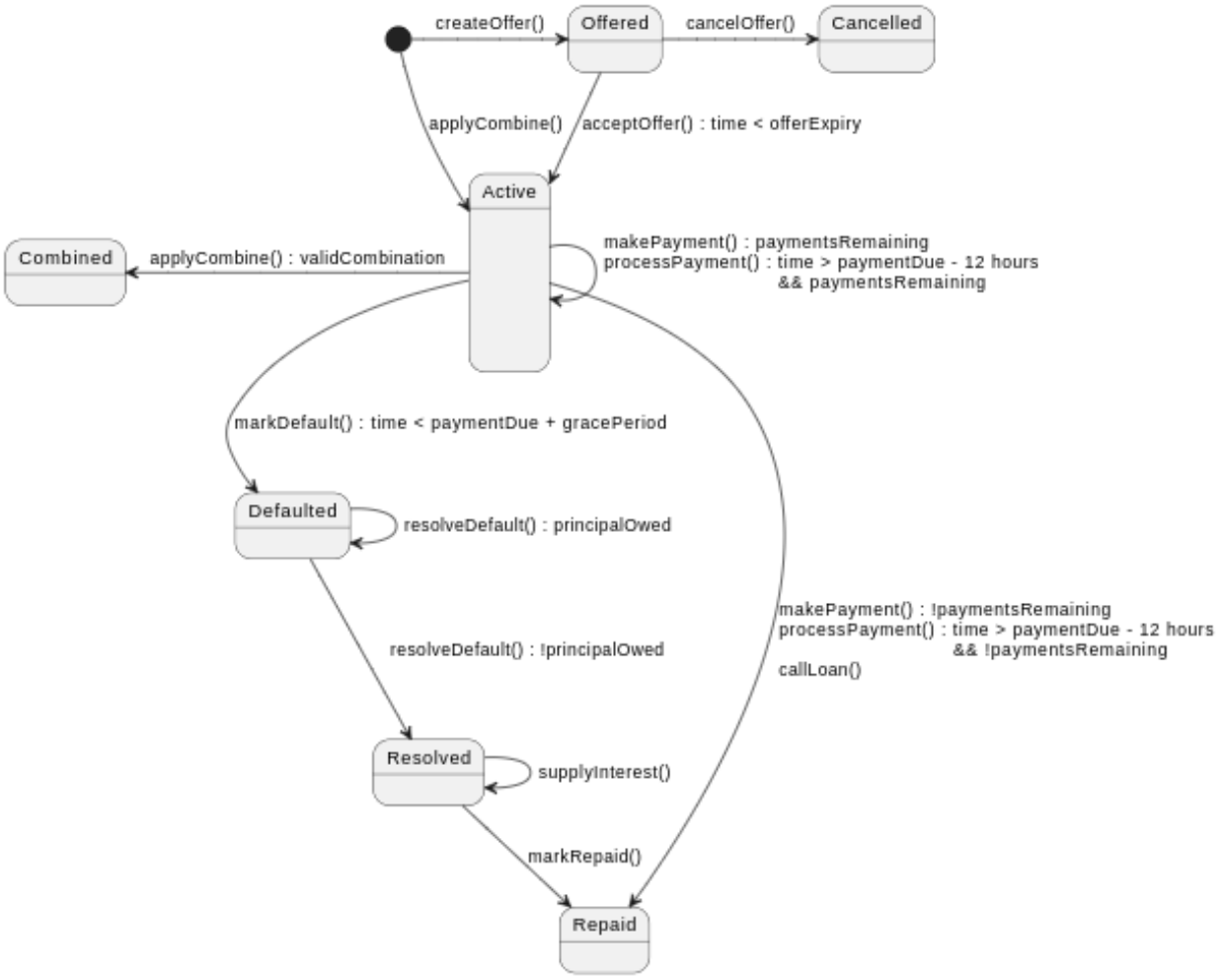
OCC Modular

The name of this locker stands for "On-Chain Credit Modular". This locker facilitates the management of loans offered by Zivoe. It contains necessary functionality that governs the life cycle of provided loans. Loans are represented by a data structure that contains the following information:

Property	**Meaning**
Borrower	The address of the borrower account
Owed principal	The total principal owed by the borrower
APR	The annualized percentage rate charged on the outstanding principal
APR Late Fee	The APR charged on the outstanding principal if payment is late
Payment Due	The timestamp (in seconds) for when the next payment is due
Remaining Payment	The number of payments remaining until the loan is repaid

Term	The number of paymentIntervals that should occur before the loan is fully repaid
Payment Interval	The interval of time between payments (in seconds)
Offer expiry	The block.timestamp at which the offer for this loan expires.
Grace period	The number of seconds a borrower has to make payment before the loan is defaulted
Payment schedule	The payment schedule of the loan, either equals 0 for a “bullet loan” or equals 1 for an “amortized loan”
Loan state	An enumeration that tracks the state of the loan.

Provided loan may go through the following phases during its life-cycle.



Each state in this life-cycle can be described as follows:

State	Meaning
Null	Default state, loan isn't offered yet
Offered	Loan offer has been created, not accepted
Canceled	Loan offer was created, then canceled prior to acceptance
Active	Loan has been accepted, is currently receiving payments
Combined	Loan was accepted, then combined with other loans while active
Defaulted	Loan has defaulted, payments were missed, grace period passed.
Resolved	Loan was accepted, then there was a default, then the full amount of principal was repaid
Repaid	Loan was accepted, and has been fully repaid

There are also two additional concepts that play an important role in the governance of loans.

- Loan schedule: Currently two different schedule options are provided as bullet and amortization loan.
    - Bullet loan: An interest-only loan, with principal repaid in full at the end.
    - Amortization loan: A principal and interest loan, with consistent payments until fully repaid.
- Loan combination: A set of loans by the same borrower can be combined as a single loan if requested. This operation is handled via a separate cycle where a combination is offered via a separate data structure and added to the list of loans as a regular loan if approved. This combination calculates the new fields of a loan as follows.

Invariant: (for each loan)

$$\texttt{owed principal + paid amount} \geq \texttt{total Debt}$$

If there are no additional fees the left and right hand sides of the inequality should be equal.

Allowed operations: `push, pull, partial pull`

Transfers:
– Accept offer: Owed principal transferred from locker to borrower
– Call loan : Interest and fees transferred from borrower to a yield locker
– Call loan : Owed principal transferred from borrower to locker owner
– Make payment : Interest and fees transferred from message sender to a yield locker
– Make payment : Owed principal transferred from message sender to locker owner
– Process payment : Interest and fees transferred from borrower to a yield locker
– Process payment : Owed principal transferred from borrower to locker owner
– Resolve default : Amount in the parameter is transferred from message sender to locker owner

– Supply interest : Amount in the parameter is transferred from message sender to locker owner

OCE ZVE

This locker handles emissions of ZVE rewards to related contracts. During the emissions some proportion of the forwarded ZVE is kept in the locker. The amount to be kept in the locker is determined by a time based function (`decay`) that determines the amount of the proportion to be kept for preservation with a resolution of seconds.

This proportion is kept in the variable `exponentialDecayPerSecond` in `RAY` precision. For instance if `exponentialDecayPerSecond=0.99999999` then the total proportion of the initial amount to be kept after 6 months will be calculated approximately %85.6 as follows:

$(0.9999999)^{15552000} \approx 0.856$

where 15552000 is the number of seconds in six months.

Calculations are done via MakerDAO's math library which is assumed to be correct.

Allowed operations: push (only ZVE can be pushed), pull, partial pull

Transfers:
– Push to locker: Amount in the parameter is transferred from locker owner to locker
– Forward emissions: Calculated amount is transferred from locker to reward contract

OCL ZVE

In this contract assets that will be provided as liquidity to the pools of Uniswap and Sushiswap are managed. This locker basically supports pushing/pulling assets (paired with ZVE) to be added to/removed from the Uniswap or Sushiswap pools and provides yield forwarding in specific time intervals (30 days) to the locker owner. During the yield forwarding a compound of the gained yields will be forwarded to the Zivoe lockers.

Allowed operations: `multi push, pull, partial pull`

Transfers:
– Push to locker multi: Amounts in the parameter are transferred from locker owner to locker

and then a calculated portion of these amounts are transferred from locker to swapper protocol
– Pull/partial from locker: Swapped amount is transferred from swapper protocol to locker and then a calculated portion of it is transferred from locker to locker owner,
– Forward yield : Swapped amount is transferred from swapper protocol to locker and then calculated portions of it are transferred from locker to yield locker and locker to locker owner.

Integrations: Uniswap or SushiSwap (`addLiquidity/removeLiquidity` functions)
– Push to locker multi: assets are added to the liquidity pool
– Pull/partial from locker: assets are removed from the liquidity pool
– Forward yield : assets are removed from the liquidity pool

OCR Modular

This locker handles the redemptions of tranche tokens to stable coins. Locker collects redemption requests in a specific data structure that keeps the following information:

Property	**Meaning**
Account	The address of the account making the request
Amount	The amount of the request
Unlocks	The timestamp after which this request may be processed
Tranche type	Boolean value that represents junior tranche if true, senior tranche otherwise

It is possible to create a request and destroy it if not processed. Processing a request involves calculating the redeemable amount by considering the current status of the redemption amounts and tranches. Each redemption is subject to a redemption fee where the collected fee is transferred to the governance.

Invariant:

```
allowed redemptions + queued redemptions = active redemptions
```

where active redemptions can be kept track by the following:

```
created request amount - (destroyed request amount - burned tokens)
```

Allowed operations: `push, pull, partial pull`

Transfers:

– Push to locker: Amount in the parameter is transferred from locker owner to locker
– Pull/partial from locker: All the balance/amount in the parameter is transferred from locker

to locker owner
– Create request : Amount in the parameter is transferred from message sender to locker
– Destroy request : Amount in the request is transferred from locker to request owner
– Process request : A calculated portion of requested amount is transferred from locker to the request owner and another portion is transferred from locker to DAO

OCT DAO

This locker is responsible for asset conversion over 1Inch integration. Assets in this locker can be swapped for another specified asset and forwarded to DAO.

Allowed operations: `push, multi push, pull, multi pull, partial pull, multi partial pull`

Transfers:
– Convert asset and forward: Whole balance of the locker is transferred to from locker to 1inch and then converted amount is transferred from the locker to DAO

Integrations: 1inch
– Convert asset and forward: Whole balance of the locker of a certain asset is converted to another

OCT YDL

This locker is responsible for asset conversion over 1Inch integration. Assets in this locker can be swapped for another specified asset and forwarded to YDL.

Allowed operations: `pull, multi pull, partial pull, multi partial pull`

Transfers:
– Convert asset and forward: Whole balance of the locker is transferred to from locker to 1inch and then converted amount is transferred from the locker to YDL

Integrations: 1inch
– Convert asset and forward: Whole balance of the locker of a certain asset are converted to YDL's asset

OCT ZVL

This locker allows Zivoe Labs to claim ZVE.

Allowed operations: `push`, `push multi`, `pull`, `multi pull`, `partial pull`, `multi partial pull`

Transfers:
– Claim: Whole balance of the locker is transferred to from locker to ZVL

OCY Convex A-B

This locker manages staking and pool operations on Convex over a set of ERC20 tokens. Push/pull operations on the locker causes tokens to be deposited and withdrawn to Convex pools. It also manages collecting rewards and forwarding them to yield lockers.

Convex A tokens: FRAX, USDC and alUSD
Convex B tokens: DAI, USDC, USDT and sUSD

Allowed operations: `push` (only allowed tokens above can be pushed), `pull`, `partial pull`

Transfers:
– Push to locker: Amount in the parameter is transferred from locker owner to locker
– Pull/partial from locker: Whole balance/Amount in the parameter is transferred from locker to locker owner
– Claim rewards: Rewards from convex are transferred from locker to OCT_YDL

Integrations: Curve and Convex
– Push to locker: Assets are added to the liquidity pool
– Pull/partial from locker: Assets are removed from the liquidity pool
– Claim rewards: Reward collection from Convex pool(s)

ZivoeSwapper

This contract provides API calls to four different 1Inch functions to address certain operations conducted by the third-parties (e.g. Uniswap, Sushiswap). This contract is used in OCT_DAO and OCT_YDL contracts to perform swapping operations from a provided asset to another asset. For OCT_YDL, the target asset is the distributed asset as yield by the locker whereas for OCT_DAO arbitrary ERC20 assets are allowed.

Within the contract, following functions are supported to be invoked over 1Inch

1Inch call	Description
swap()	Execute over 1Inch supported sources
uniswapV3Swap()	Execute over Uniswap V3 or compatible pools
unoswap()	Execute over Uniswap V2 or compatible pools
fillOrderRFQ()	Execute over limit orders

For each of these functionality, the contract supports validation to be performed over the byte-array data provided as a parameter. The `convertAsset` function provided by the contract takes additional arguments to the byte-array function call information and according to the function call encoded in the byte-array an internal validation function is invoked. The additional arguments that contain the information of swapping token pair addresses and the amount to be swapped is forwarded to the internal validation function and the validation is done by extracting out, decoding and comparing the related sub-portions of the byte-array argument.

After the validation a low level call is performed over the 1Inch router using the provided byte-array.

OCY USD

This locker is responsible for managing the OUSD assets and the yield beared by OUSD token. The amount of owned OUSD tokens may change due to the price fluctuations if rebasing is enabled. This contract keeps track of the original amount of OUSD tokens kept in the contract and forwards the yields by OUSD (if any) to the OCT_YDL locker.

Allowed operations: `push, pull, partial pull`

Transfers:
– Push to locker: Amount in the parameter is transferred from locker owner to locker
– Pull/partial from locker: Whole balance/Amount in the parameter is transferred from locker to locker owner
– Forward yield : Excess amount of OUSD (if present) is transferred from locker to OCT_YDL

Findings

A01: Yield generated in `OCY_OUSD` may not be distributed over token holders

[Severity: Medium | Difficulty: High | Category: Security]

In `OCY_OUSD` contract, it is possible to call the `pullFromLocker` function and transfer the accrued yield even though there exists a separate `forwardYield` function which transfers this yield to `OCT_YDL`.

Scenario

1. `ZivoeDAO` pushes capital into `OCY_OUSD`
2. After some time, yield is generated, but the function distributeYield is not called, therefore the yield is not transferred into `OCY_YDL`
3. `ZivoeDAO` pulls capital from the `OCY_OUSD` and the yield generated is transferred into `ZivoeDAO` instead of `OCY_YDL`
4. Owners of tranche tokens do not receive their portion of yield.

Recommendation

Although the situation is reversible by pushing the yield back to `OCY_OUSD`, or directly into `OCY_YDL`, we recommend to distribute the yield before transferring the OUSD in the pull functions.

Status

Addressed in commits `001694b8ff324722549aac0f80fb65d18593ef94`, `a414ff79d7e24d590350fe31b40976aaf28e5bab`, `431d69f71e4c6bceca0780e06ae8c976068aa133` and `a2de0ecd67e2d951106a2eea22c166183f4fb534`.

A02: `updateOCTYDL` functions do not check for `address(0)`

[Severity: Medium | Difficulty: High | Category: Input Validation]

The function `updateOCTYDL` in contracts `OCC_Modular`, `OCL_ZVE`, `OCY_Convex_A`, `OCY_Convex_B` and `OCY_OUSD` update the address of `OCT_YDL` contract. These contracts transfer the yield generated to the `OCT_YDL` contract, which is responsible for distributing the yield generated to the tranche token holders.

Scenario

1. `ZivoeDAO` pushes capital into `OCL_ZVE` (similarly for `OCY_Convex_A`, `OCY_Convex_B` and `OCY_OUSD`)
2. `ZivoeDAO` calls `updateOCTYDL(address(0))`
3. When the function `forwardYield` is called, if the `pairAsset` token does not revert on transfer for `address(0)` the generated yield is burned

Recommendation

Add checks for `address(0)` in `updateOCTYDL` functions.

Status

Addressed in commit [d701bd7838586c9a5be4260ddc459e4295746648](d701bd7838586c9a5be4260ddc459e4295746648).

A03: Token redemption transfers redemption fee to the owner

[Severity: High | Difficulty: Low | Category: Functional Correctness]

In the `OCR_Modular` contract, the function `processRequest` processes a redemption request, where the owner of tranche tokens redeems his tokens for some `stablecoin`. When redeeming the tokens, a redemption fee should be transferred to `DAO`. However, the function is actually transferring the fee to the token holder and the remaining to `DAO`.

```
IERC20(stablecoin).safeTransfer(requests[id].account, redeemAmount *
redemptionsFeeBIPS / BIPS);
IERC20(stablecoin).safeTransfer(IZivoeGlobals_OCR(GBL).DAO(), redeemAmount * (BIPS
- redemptionsFeeBIPS) / BIPS);
```

Besides, the `redemptionsFeeBIPS` should be less than 20%, as enforced by the function `updateRedemptionsFeeBIPS`, which means that the token holder receives up to 20% of the `redeemAmount` whereas the `DAO` receives the 80% or more.

Recommendation

Switch the transfer amounts between `requests[id].account` and `DAO`.

Status

Addressed in commit 842a2d2c66e7b0db238793d8899c5aa97a45ddeb.

A04: Minimum threshold values for liquidity operations in pool integrations

[Severity: High | Difficulty: Low | Category: Integration]

In the `OCL_ZVE` contract, when adding liquidity to Uniswap/Sushiswap pool minimum bound values are either set the same as the desired values.

Related call is as follows:

```
(uint256 depositedPairAsset, uint256 depositedZVE, uint256 minted) =
IRouter_OCL_ZVE(router).addLiquidity(
     pairAsset,
     IZivoeGlobals_OCL_ZVE(GBL).ZVE(),
     IERC20(pairAsset).balanceOf(address(this)),
     IERC20(IZivoeGlobals_OCL_ZVE(GBL).ZVE()).balanceOf(address(this)),
     IERC20(pairAsset).balanceOf(address(this)),
     IERC20(IZivoeGlobals_OCL_ZVE(GBL).ZVE()).balanceOf(address(this)),
     address(this), block.timestamp + 14 days
 );
```

This call expects the minimum rate to be exactly the same as the desired rate which would result in a revert probably all the time.

Recommendation

Provide an adjustable minimum bound for these calls so that a meaningful margin is set when adding liquidity to pools.

Status

Addressed in commit [c6a13971c1ed2c24e2fa2640833e4a8e9146d404](c6a13971c1ed2c24e2fa2640833e4a8e9146d404).

A05: `_forwardYield()` uses `compoundingRateBIPS` incorrectly

[Severity: Low | Difficulty: Low | Category: Functional Correctness]

In the `OCL_ZVE` contract, the function `_forwardYield` computes the amount of tokens that can be claimed as yield generated during a period of time (30 days). It is expected that the function compounds a certain percentage of the generated yield (`compoundingRateBIPS`) and distributes the remaining. However, the amount of liquidity tokens that will be burned to claim the yield to be distributed is computed as follows:

```
uint256 lpBurnable = (amount - basis) * lp / amount * compoundingRateBIPS / BIPS;
```

According to this calculation, the `compoundingRateBIPS` is actually the percentage of yield being distributed, which does not go along with the documentation in the `updateCompoundingRateBIPS`:

```
/// @dev    A value of 2,000 represents 20% of the earnings stays in this
contract, compounding.
```

Recommendation

Calculate the `lpBurnable` amount as:

```
uint256 lpBurnable = (amount - basis) * lp / amount * (BIPS -
compoundingRateBIPS) / BIPS;
```

Status

Addressed in commit [c35d3e114cbf9376104d325178f986359b0fa4ce](c35d3e114cbf9376104d325178f986359b0fa4ce).

A06: `applyCombine()` has unnecessary modulo calculation

[Severity: High | Difficulty: Low | Category: Functional Correctness]

In the `OCC_Modular` contract, the function applyCombine combines multiple loans of the same borrower into a single loan. In order to do that, the `principalOwed` - `notional` - and the `APR` of the combined loan are calculated as follows:

```
notional += loans[loanID].principalOwed;
APR += loans[loanID].principalOwed * loans[loanID].APR;
```

Where `loanId` stands for each loan to be combined. Then the `APR` is calculated as:

```
APR = APR / notional % 10000;
```

Scenario

If the sum `loans[loanID].principalOwed * loans[loanID].APR` of the various loans to be combined is equal to 100%, then the `APR` of the combined loan will be 0.

Recommendation

Remove the modulo calculation from the APR calculation.

Status

Addressed in commit [c4b4c92a389f2d31e923884cd4e53e66f4ef3ead](c4b4c92a389f2d31e923884cd4e53e66f4ef3ead).

A07: `resolveDefault()` lacks `nonReentrant` modifier

[Severity: Low | Difficulty: High | Category: Security]

In the `OCC_Modular` contract, all the functions interacting with the external contract have the nonReentrant modifier, except for the `resolveDefault` function. For this to be exploitable, either the `stablecoin` or the `GBL` contracts would have to be malicious, which is very unlikely to happen. But, if that was the case they would have the power to mark as many loans as they would like as resolved.

Recommendation

Add the `nonReentrant` modifier to the `resolveDefault` function.

Status

Addressed in commit [d96a3cd71e6b21c98e176bd738ee78d49d4a27a5](d96a3cd71e6b21c98e176bd738ee78d49d4a27a5).

A08: `exponentialDecayPerSecond` does not have a range

[Severity: Medium | Difficulty: High | Category: Input Validation]

In the `OCE_ZVE` contract, The `exponentialDecayPerSecond` variable will determine how many emissions per second are forward to `ZivoeRewards` contracts. The governance can set the `exponentialDecayPerSecond` variable to modify the decay emissions schedule. However, there are no constraints on the value being updated, making it possible to decay all the `ZVE` in a very short period of time.

Recommendation

Add a lower bound for the new value `_exponentialDecayPerSecond` in the function `updateExponentialDecayPerSecond`.

Status

Addressed in commit [`b5fe0dea3874e38dbcb416ff3064c096980cd6f3`].

A09: Precision of `portion` in `processRequest` is not correct

[Severity: High | Difficulty: Low | Category: Precision]

In the `OCR_Modular` contract, during the calculation of the amount of tranche tokens to be burned, the proportion of the balance of stable coins owned by the locker to the amount of total redemptions is used as follows

```
uint256 totalRedemptions = redemptionsAllowedSenior *
                        (BIPS - epochDiscountSenior) +
                        (redemptionsAllowedJunior * (BIPS - epochDiscountJunior));
uint256 portion = (IERC20(stablecoin).balanceOf(address(this)) * RAY
                  / totalRedemptions) / 10**23;
```

However, this calculation

1. do not consider the precision difference between the stable coin and the redemptions and,
2. do not produce a result in BIPS precision if precision in 1 are the same

Scenario

Let's assume:

- redemptionsAllowed and stablecoin uses 18 decimals
- redemptionsAllowedSenior is 10^{17} (0.1)
- redemptionsAllowedJunior is also 10^{17}
- epochDiscountSenior is 1000 (10%)
- epochDiscountSenior is also 1000
- stablecoin balance is 10^{17} (0.1)

For the values above, total redemptions is 0.18 and the portion is 0.1 / 0.18 which is 55.55 and in turn in BIPS precision 5555. However it is being calculated as

```
totalRedemptions = 10**17 * (10**4 - 10**3) + 10**17 * (10**4 - 10**3)
                 = 10**17 * (9*10**3) + 10**17 * (9*10**3)
                 = 18 * 10**20

portion = 10**17 * 10**27 / 18*10**20 / 10**23
        = 0
```

Recommendation

1. It should be ensured that the stable coin balance and the redemption values are in the same precision before the calculation
2. Assuming item 1 is ensured and the rest of the code is not modified, denominator in the portion calculation should be 10^{19}

Status

Addressed in commit [842a2d2c66e7b0db238793d8899c5aa97a45ddeb]

Informative Findings

B01: Inconsistent behavior in `pullFromLocker` functions

[Severity: - | Difficulty: - | Category: -]

In the `OCY_Convex_B` contract, pull functions call `claimRewards` before the pull operations are done. However, in `OCY_Convex_A`, a similar call is not performed and the invocation of `claimRewards` is left to user initiative.

Recommendation

Handling of `reward/yield` harvesting can be performed more consistently across the `OCY` contracts.

Status

Addressed in commit `001694b8ff324722549aac0f80fb65d18593ef94`.

B02: `tickEpoch` can be more gas efficient if done via a `loop`

[Severity: - | Difficulty: - | Category: Gas Optimization]

The function `tickEpoch` in the `OCR_Modular` contract is recursive.

Recommendation

It may be possible to mitigate additional gas consumption if a `loop` is used.

Status

Addressed in commit `96dcaa5d969bd6ed80d79a7a3e930ff82e80795b`.

B03: Precomputing values in `_forwardEmissions`

[Severity: - | Difficulty: - | Category: Gas Optimization]

The function `_forwardEmissions` in the `OCE_ZVE` contract repeatedly computes the value `amount * distributionRatioBIPS[i] / BIPS` for `i` $\in \{0, 1, 2\}$.

Recommendation

Precompute and reuse the value.

Status

Addressed in commit 962c5c42ecd85a8189321ccee5346261c4ac7bd1.

B04: outdated documentation in `processPayment` function

[Severity: - | Difficulty: - | Category: Documentation]

In the `OCC_Modular` contract, the documentation relative to the function `processPayment` is outdated.

```
/// @dev    Anyone is allowed to process a payment, it will take from "borrower".
/// @dev    Only allowed to call this if block.timestamp > paymentDueBy.
```

But, in fact, only `underwriter` or `keepers` are allowed to call this function. Besides, the function can be called up to 12 hours before the `paymentDueBy`.

Recommendation

Update the documentation accordingly.

Status

Addressed in commit [a680b3d4bd56eb8864c3bbf28125ecd7ca5988a7](a680b3d4bd56eb8864c3bbf28125ecd7ca5988a7).

B05: Computed value not used

[Severity: - | Difficulty: - | Category: Gas Optimization]

In the `OCC_Modular` contract, the function `amountOwed` has the following signature:

```
function amountOwed(uint256 id) public view returns (
     uint256 principal, uint256 interest, uint256 lateFee, uint256 total )
```

where the returning value `total`, calculated as `total = principal + interest + lateFee`, is not used anywhere.

Recommendation

The `total` returning value and its computation can be deleted.

Status

Zivoe decided to leave this in as they consume the value in a subgraph currently.

B06: Reused external calls can be kept in local variables

[Severity: - | Difficulty: - | Category: Gas Optimization]

Functions such as `OCL_ZVE::pushToLockerMulti()` makes multiple external calls throughout the function to query the same value.

Recommendation

Performing these calls once and storing the value in a local variable and reusing this variable can save gas.

Status

Addressed in commits [b4f2b60aca746db36b5dd8302b15cad797cc40fe](), [09be9cabab914e5b1667ccefcc391a362d376b65](), [4f5c69adffa8b1b0b33713175fc3b9c4999969c8](), [125cf5ca3ac5a87e8e440f2c8f5df28248635ac3](), [b4f2b60aca746db36b5dd8302b15cad797cc40fe]().

B07: Check that amount to be transferred is greater than 0

[Severity: - | Difficulty: - | Category: Gas Optimization]

The functions `makePayment` and `processPayment`, in the `OCC_Modular` contract, calculate the `principalOwed` that must be paid and transferred to the `owner`. However, for bullet loans, the `principalOwed` is always 0 except for the last payment.

Recommendation

Checking that `principalOwed > 0` before the transfer would avoid spending gas on an empty transfer.

Status

Addressed in commit [3556c664e3d18a0128f3202db24e496489794bb6](3556c664e3d18a0128f3202db24e496489794bb6).

B08: Minimum expected values for liquidity operations in pool integrations

[Severity: High | Difficulty: Low | Category: Integration]

When removing liquidity from integrated pools minimum values are set to 0.

An example from `OCL_ZVE` to remove liquidity from a pool is as follows:

```
(uint256 claimedPairAsset, uint256 claimedZVE) =
IRouter_OCL_ZVE(router).removeLiquidity(
     pairAsset, IZivoeGlobals_OCL_ZVE(GBL).ZVE(), preBalLPToken,
     0, 0, address(this), block.timestamp + 14 days
 );
```

Which can result in an undesirably small amount to be obtained by the call.

More instances are present for `OCY_Convex` lockers.

Recommendation

Provide an adjustable minimum rate for these calls so that potential erratic behavior(s) by the integrated systems can be prevented.

Status

Addressed in commit [959ced49746801ddfb8145c3cc387056d9dfeb85](), [591e1dee200ecd060c944ef0070dddf7b9598a32](), [125cf5ca3ac5a87e8e440f2c8f5df28248635ac3]().

B09: `processRequest` does not revert if `redeemAmount` is 0

[Severity: - | Difficulty: - | Category: Gas Optimization, Usability]

The function `processRequest` in `OCR_Modular` does not revert if the amount to be redeemed is 0. This leads to unnecessary expenditure of gas on an empty transfer. Besides, the user does not receive feedback that his redemption actually was not enough to redeem any tokens.

Recommendation

Revert if `redeemAmount` is equal to 0.

Status

Addressed in commit `842a2d2c66e7b0db238793d8899c5aa97a45ddeb`.

B09: Incorrect naming of contract

[Severity: - | Difficulty: - | Category: -]

The name of the contract in file `OCT_ZVL.sol` is `OCT_DAO`.

Recommendation

Contract name should be set to the intended name `OCT_ZVL`.

Status

Addressed in commit `03af61a5f3e21e53dbf232fca9282e5c0d104895`